Zostań ekspertem od mikrokontrolerów PIC!

Mikrokontrolery PIC

w praktycznych zastosowaniach

Paweł Borkowski

- Poznaj architekturę mikrokontrolerów PIC
- Naucz się programować je w asemblerze i języku C
- Dowiedz się, jak sterować urządzeniami zewnętrznymi

Helion

Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Maciej Pasek

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?mipicp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
ftp://ftp.helion.pl/przyklady/mipicp.zip

ISBN: 978-83-246-3721-8

Printed in Poland.

Spis treści

Rozdział 2.

PIC16F877A

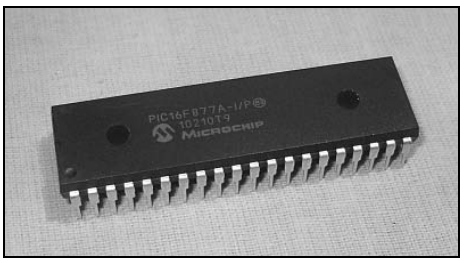
Obsługa diod LED. Obsługa wyświetlaczy LED i alfanumerycznych wyświetlaczy LCD. Obsługa serwomechanizmów

2.1. Język C

Obsługa diod LED

Nasz pierwszy mikrokontroler PIC prezentuje się tak jak na rysunku 2.1.

Rysunek 2.1. *Mikrokontroler PIC16F877A*



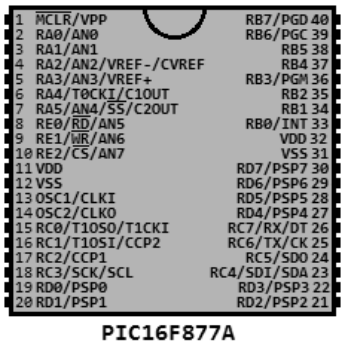
Skromny, ale odważny. Ma 40 bohaterskich nóżek, które nie zawahają się uczynić tego, do czego je zaprogramujemy. Proszę mi wybaczyć tę nieścisłość. Programujemy oczywiście wyprowadzenia, których fizycznym przedłużeniem są owe nóżki. Mądrzy ludzie powiadają, że uroda to nie wszystko. Więc czymże ten heros może się pochwalić, jeśli chodzi o jego wnętrze? Spójrzmy na rysunek 2.2, który pochodzi z dokumentacji mikrokontrolera.

Device	Program Memory		Data SRAM (Bytes)	EEPROM (Bytes)	I/O	10-bit A/D (ch)	CCP (PWM)	MSSP		USART	Timers 8/16-bit	Comparators
	Bytes	# Single Word Instructions						SPI	Master I^2C			
PIC16F873A	7.2K	4096	192	128	22	5	2	Yes	Yes	Yes	2/1	2
PIC16F874A	7.2K	4096	192	128	33	8	2	Yes	Yes	Yes	2/1	2
PIC16F876A	14.3K	8192	368	256	22	5	2	Yes	Yes	Yes	2/1	2
PIC16F877A	14.3K	8192	368	256	33	8	2	Yes	Yes	Yes	2/1	2

Rysunek 2.2. *Zestawienie wyposażenia mikrokontrolerów rodziny 16F877A. Rysunek pochodzi z dokumentacji mikrokontrolera (DS39582B), s. 1*

SPI? Yes! Master I^2C? Yes! USART? Yes! Czyli trzy razy „yes"! Zaczyna się nie najgorzej. Na razie interesować nas będzie blok oznaczony I/O, czyli cyfrowy interfejs wejścia/wyjścia. Właśnie pod tym hasłem kryją się ulubione przez nas terminy *port* i *linie portu*. Wszystko zmierza ku jednemu — aby pojawił się rysunek trzeci, tym razem z opisem wyprowadzeń mikrokontrolera (patrz rysunek 2.3).

Rysunek 2.3. *Rozmieszczenie wyprowadzeń mikrokontrolera PIC16F877A*

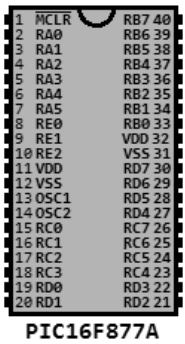


Porównajmy rysunki 2.1 i 2.3. Czy rzeczywiście mikrokontroler jest tak szeroki, jak widać na rysunku 2.3? Raczej nie. Poszerzyła go konieczność zmieszczenia wszystkich funkcji wyprowadzeń. Weźmy taką nazwę: RA3/AN3/VREF+. Oznacza ona, że wyprowadzenie o numerze 5 pełni trzy alternatywne funkcje: linii portu A (RA3), wejścia analogowego (AN3) i czegoś tam jeszcze (VREF+). Przełączanie między tymi funkcjami wiąże się z ustawianiem odpowiednich wartości na odpowiednich rejestrach. Nie jest to trudne, ale wymaga od nas dobrej znajomości dokumentacji programowanego układu. Na ogół przy pierwszych programach, którymi najczęściej są ekscesy z diodami, alternatywne funkcje wyprowadzeń nie powinny nas martwić. Koniecznie jednak sprawdźmy, czy linia, do której podłączyliśmy diodę, jest domyślnie linią I/O. Może się bowiem okazać, co jest zjawiskiem nagminnym w mikrokontrolerach PIC, że domyślnie jest to wejście analogowe.

Wróćmy jednak do prapoczątków. Musimy bowiem założyć, że wśród Czytelników są także mikrokontrolerowi nowicjusze. Na razie wiemy tyle: mikrokontroler to kostka z nóżkami, inaczej wyprowadzeniami. Nazwy wyprowadzeń wskazują na ich funkcję. A jeśli wyprowadzenie składa się z kilku nazw oddzielonych ukośnikiem, oznacza to, że wyprowadzenie pełni kilka funkcji. Zapowiedziałem, że mamy się tym nie martwić. Więc się nie martwmy. My też pełnimy kilka alternatywnych funkcji, co nas wcale nie smuci. Na dodatek nie witamy się, podając ich pełną listę: „Cześć, Jacku

Michale Czarnecki, mężu Grażyny, ojcu Żanety i Edgara, sprzedawco w osiedlowym sklepie spożywczym, teraz przechodzę do informacji niejawnych…". A więc, powtarzam, tak się nie witamy. Raczej powiemy sobie: „Cześć, Jacku" i po kłopocie. Taka sama zasada obowiązuje w przypadku opisu wyprowadzeń mikrokontrolera. Najwygodniej jest przedstawić układ, w którym z wielu funkcji wyprowadzeń pozostawiamy jedynie nazwy linii portu. Nazwę dodatkowej funkcji będziemy umieszczać wtedy, gdy będziemy z niej korzystać (patrz rysunek 2.4).

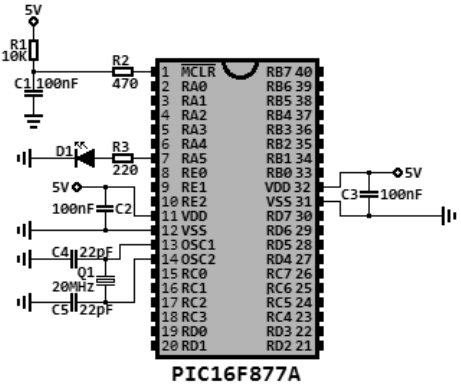
Rysunek 2.4. *Rozmieszczenie wyprowadzeń mikrokontrolera PIC16F877A w wersji uproszczonej*



Rysunek mikrokontrolera wygląda teraz o wiele lepiej. W ten oto sposób rozwiązaliśmy pierwszy problem, pozostało ich jeszcze 728. Zajmiemy się nimi po kolei.

Tradycja każe, by pierwszy mikrokontrolerowy program dotyczył układu z diodą LED[1], przy czym dioda ma świecić, a nawet — ku zgorszeniu wszystkich — mrugać. Schemat naszego pierwszego układu z diodą LED został przedstawiony na rysunku 2.5.

Rysunek 2.5. *Schemat układu z mikrokontrolerem PIC16F877A i podłączoną do niego diodą LED*

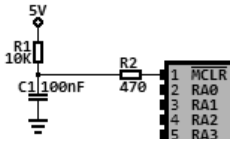


Mikrokontrolerowi wyjadacze teraz ziewają, lecz nowicjuszy opanował lekki strach. Czy układ rzeczywiście musi tak wyglądać? Dlaczego przy wyprowadzeniu MCLR jest to, co jest? Po co te kondensatory przy liniach zasilających? Odpowiadam na pytanie

[1] W literaturze dotyczącej mikrokontrolerów pojęcie *układu* często używane jest w dwóch znaczeniach: pierwszym, dotyczącym mikrokontrolera, i dodatkowo drugim, odnoszącym się do obwodów elektrycznych, w których ów mikrokontroler występuje. Najczęściej z kontekstu łatwo się domyślić, o które znaczenie chodzi. Cóż, mikrokontrolery to taka dziedzina wiedzy, w której nawet pojęcia pełnią alternatywne funkcje.

pierwsze: nie, układ nie musi tak wyglądać. Rysunek 2.5 przedstawia schemat mega-poprawny, zgodny z kanonem, mówiąc wprost — grzeczny. Układ został przygotowany na wszelkiego typu niespodzianki, jak wybuch wulkanu, plamy na Słońcu itp. Rozważmy elementy schematu po kolei. Zacznijmy od układu RESET (rysunek 2.6).

Rysunek 2.6.
Układ RESET

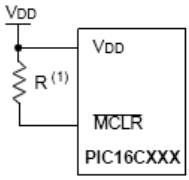


Jest wielce prawdopodobne, szanowny Czytelniku, że widziałeś lub zobaczysz wiele podobnych albo zupełnie różnych układów RESET. Nic dziwnego. Chodzi wszakże o to, by obwód działał w charakterystyczny sposób, a osiągnąć to można na wiele sposobów. Układy PIC resetowane są po podaniu niskiego poziomu na wejście MCLR. Sygnał musi być odpowiednio długi, gdyż krótkie spadki napięcia na linii MCLR są ignorowane dzięki specjalnym filtrom szumu. Rozpoznanie sygnału RESET wymusza przyjęcie przez rejestry wartości początkowych. Jakich? To można sprawdzić w dokumentacji. Dodatkowo istnieje wiele źródeł resetujących mikrokontroler:

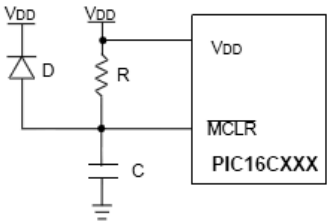
1. RESET po włączeniu zasilania (Power-on Reset **POR**);
2. tak zwany normalny RESET, wymuszony sygnałem niskim w trakcie pracy układu;
3. RESET po przejściu w stan uśpienia;
4. RESET wywołany sygnałem modułu nadzorującego watchdog;
5. RESET wywołany chwilowym spadkiem napięcia.

Do każdego z wymienionych punktów można zaprojektować specyficzny układ RESET. Nas interesuje jedynie zbudowanie obwodu zapewniającego dostarczenie do mikrokontrolera sygnału zerującego po włączeniu zasilania (POR). Natomiast w trakcie pracy mikrokontrolera na linię MCLR powinien być dostarczany stabilny sygnał zapobiegający zerowaniu układu. Z tego wszystkiego powinniśmy wysnuć taki wniosek: linia MCLR nie powinna *wisieć w powietrzu*, lecz na czas pracy układu powinien być na niej wymuszony stan wysoki. W jaki sposób to osiągnąć — oto całe zagadnienie, nad którym właśnie mamy przyjemność się pochylać. Rysunek 2.6 prezentuje jedno z możliwych rozwiązań. Prześledźmy najprostsze rozwiązania występujące w literaturze. Rysunki 2.7 i 2.8 przedstawiają dwie propozycje pochodzące z dokumentacji PICmicro MID-RANGE MCU FAMILY.

Rysunek 2.7.
Układ POR z opcjonalnym rezystorem. Rysunek pochodzi z dokumentacji PICmicro MID-RANGE MCU FAMILY (DS31003A), s. 3 – 4



Rysunek 2.8.
Układ POR przedłużający czas włączenia zasilania. Rysunek pochodzi z dokumentacji PICmicro MID-RANGE MCU FAMILY (DS31003A), s. 3 – 4

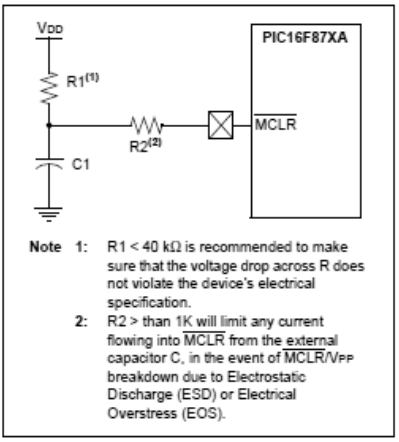


W przypadku obwodu zaprezentowanego na rysunku 2.7 rezystor może wystąpić, ale nie musi. Czyli, prawdę mówiąc, linię MCLR możemy bezpośrednio podłączyć do źródła napięcia. Jak twierdzi dokumentacja, czas załączenia układu będzie w tym przypadku wystarczający do wygenerowania sygnału RESET. Nieco odmiennego zdania jest dokumentacja mikrokontrolerów rodziny PIC16F8XA. Na stronie 148 tej dokumentacji możemy przeczytać:

> *[...] Microchip recommends that the MCLR pin no longer be tied directly to VDD.*
>
> (DS39582B, s. 148)

I dla odmiany proponowany jest tu obwód jeszcze innej postaci (patrz rysunek 2.9).

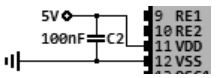
Rysunek 2.9.
Obwód MCLR. Rysunek pochodzi z dokumentacji mikrokontrolerów rodziny PIC16F8XA (DS39582B), s. 148



Podobieństwo rysunków 2.6 i 2.9 nie jest przypadkowe. Tyle na ten temat.

Drugą rzeczą związaną ze schematem z rysunku 2.5, o której wypada wspomnieć, jest postać obwodu zasilania (patrz rysunek 2.10).

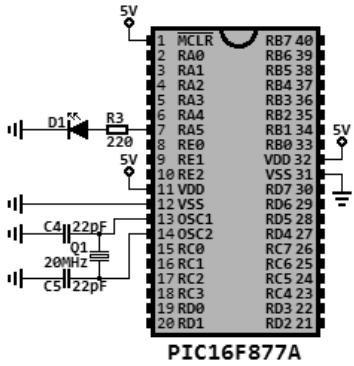
Rysunek 2.10.
Obwód zasilania mikrokontrolera PIC16F877A



Rolą dołączonego kondensatora jest eliminowanie zakłóceń o wysokiej częstotliwości (słynne *odsprzęganie zasilania*).

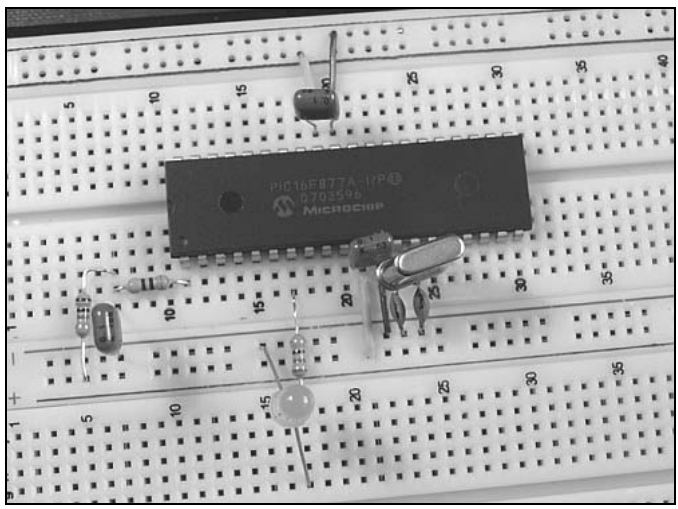
Nie skupilibyśmy się na tych elementach, gdyby to nie miało sensu. Chodzi o to, że jeśli brak nam ochoty lub wystarczającej liczby części, schemat z rysunku 2.5 możemy uprościć do postaci z rysunku 2.11.

Rysunek 2.11. *Schemat układu z mikrokontrolerem PIC16F877A i podłączoną do niego diodą LED (wersja oszczędna)*



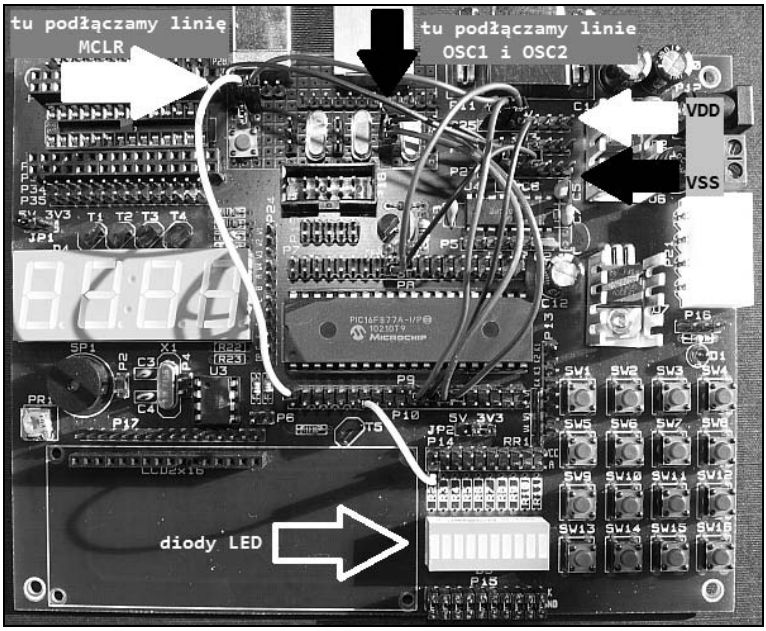
Pamiętajmy jednak, że poprawniejsza jest wersja układu z rysunku 2.5. Jej realizację na płytce stykowej przedstawia rysunek 2.12.

Rysunek 2.12. *Realizacja schematu z rysunku 2.5 na płytce stykowej*



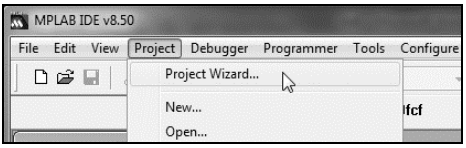
Rysunek 2.13 przedstawia układ z mikrokontrolerem PIC16F877A zrealizowany na płytce edukacyjnej ARE.

Na rysunku 2.13 został zaznaczony układ RESET, gdzie podłączamy linię MCLR. Linię MCLR podłączamy do środkowego z trzech pinów, masę podłączamy z lewej, a VDD z prawej strony pinu linii MCLR (miejmy nadzieję, że rozkład pinów się nie zmieni). Linie OSC1 i OSC2 podłączamy do skrajnych pinów stojących obok oscylatora 20 MHz. Do środkowego pinu podłączamy masę. Jeśli chodzi o źródło zasilania, to napięcie 5 V jest wyprowadzone na listwę pinów P25, a masa egzystuje sobie na listwie P27.



Rysunek 2.13. *Realizacja schematu z rysunku 2.5 na płytce edukacyjnej ARE (nie zostały umieszczone kondensatory C2 i C3)*

Czas najwyższy zacząć pisać program[2]. Uruchamiamy MPLAB IDE. By utworzyć nowy projekt, z menu wybieramy *Project/Project Wizard...* (patrz rysunek 2.14).



Rysunek 2.14. *Zakładka tworzenia nowego projektu*

Pojawi się okno tworzenia nowego projektu, w którym klikamy *Dalej*. W oknie drugim, a jak twierdzi kompilator, w kroku pierwszym, będziemy mieli za zadanie wybrać interesujący nas mikrokontroler. Ponieważ lista jest długa, zarezerwujmy sobie wolne popołudnie, gdyż czeka nas niemała przygoda. Najważniejsze, abyśmy znaleźli nazwę naszego mikrokontrolera, czyli PIC16F877A (patrz rysunek 2.15).

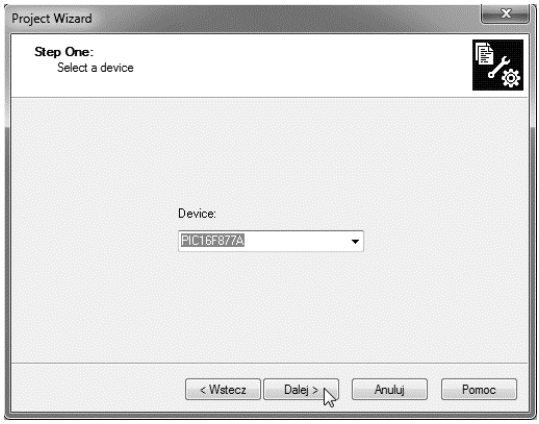
W drugim kroku konfiguracji projektu wybieramy kompilator. Jak już wiemy po fascynującej lekturze rozdziału pierwszego, kompilatorem układów 8-bitowych jest HI-TECH. Jeśli nie możemy wybrać go z listy, to znaczy, że zapomnieliśmy go zainstalować. Po naprawieniu tego małego przeoczenia na pewno uda nam się znaleźć punkt *HI-TECH Universal ToolSuite* (patrz rysunek 2.16).

[2] Jeśli jest na coś najwyższy czas, to na taką oto prawdę życiową:

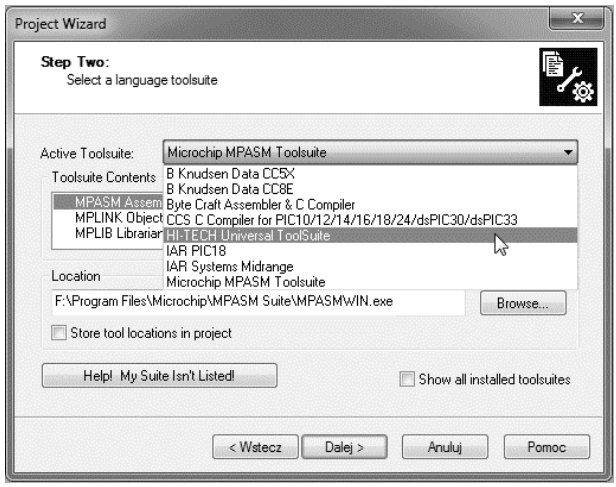
Z cyklu *Rozterki gracza futbolowego*

Nie wykonam tego kopa.
Ciągle śmieje mi się stopa.

Rysunek 2.15. *Okno wyboru mikrokontrolera*



Rysunek 2.16. *Okno wyboru kompilatora*



Trzeci krok to podanie lokalizacji i nazwy projektu. Ja wpisałem *R02_Prog01_C_*↪*PIC16F877A*, bo lubię nazwy łatwe i nieskomplikowane[3]. Człon *R02_* oznacza u mnie, że projekt jest opisywany w drugim rozdziale książki, *Prog01_* mówi, że jest to pierwszy projekt tego rozdziału, na dodatek napisany w języku C (człon *C_*).

W czwartym kroku budowania projektu zostaniemy poproszeni o wskazanie plików, które już na wstępie mają być dołączone do projektu. Prawdopodobnie niczego nie będziemy chcieli dołączyć, dlatego klikamy *Dalej*, a następnie *Zakończ*. W oknie roboczym projektu zobaczymy jego strukturę (patrz rysunek 2.17).

[3] A co się tyczy nieskomplikowanego życia, to mam przygotowaną taką prawdę życiową:

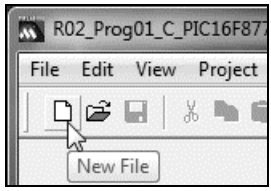
Z cyklu *Rozterki ślimaka*

Gdy jadłem sałatę — nic się nie działo.
Spokojnie wyjadłem ziarna gorczycy.
I właśnie wtedy wszystkim się zachciało
Mnie przegnać, gdy szedłem do ślimaczycy.

Rysunek 2.17. *Struktura nowo utworzonego projektu*

Kod będziemy wpisywać w pliku źródłowym, który musimy utworzyć. W tym celu klikamy ikonę *New File* (patrz rysunek 2.18).

Rysunek 2.18. *Ikona utworzenia nowego pliku*



Otworzy się okno edytora, w którym wreszcie możemy zacząć pisać program. W pierwszym wierszu musimy poinformować kompilator o chęci skorzystania z zasobów kompilatora.

```
#include <htc.h>
```

Bez tego nazwy własne, takie jak `PORTA`, `TRISA`, nie będą rozpoznawalne. Następnie ustawimy bity konfiguracyjne. Do tego służy makro `__CONFIG`.

```
//oscylator szybszy od 10 MHz (FOSC_HS)
//watchdog wyłączony (WDTE_OFF)
//wyłączone LVP (Low-Voltage ICSP Programming) (LVP_OFF)
__CONFIG(FOSC_HS & WDTE_OFF & LVP_OFF);
```

Lista argumentów makra zawiera tylko różnice w stosunku do domyślnej konfiguracji mikrokontrolera. Listę masek dostępnych dla mikrokontrolera PIC16F877A znajdziemy w zasobach kompilatora HI-TECH w pliku *pic16f877a.h*. A jeśli nie chciałoby nam się szukać, przytoczę interesujący nas fragment tego pliku. Nie należy go czytać, a jedynie z okrzykiem „E tam!" przerzucić stronę.

Listing pic16f877a.h *(fragment)*

```
//
//Configuration mask definitions
//

//Config Register: CONFIG
#define CONFIG                  0x2007
//Oscillator Selection bits
```

```
//RC oscillator
#define FOSC_EXTRC          0xFFFF
//HS oscillator
#define FOSC_HS             0xFFFE
//XT oscillator
#define FOSC_XT             0xFFFD
//LP oscillator
#define FOSC_LP             0xFFFC
//Watchdog Timer Enable bit
//WDT enabled
#define WDTE_ON             0xFFFF
//WDT disabled
#define WDTE_OFF            0xFFFB
//Power-up Timer Enable bit
//PWRT disabled
#define PWRTE_OFF           0xFFFF
//PWRT enabled
#define PWRTE_ON            0xFFF7
//Brown-out Reset Enable bit
//BOR enabled
#define BOREN_ON            0xFFFF
//BOR disabled
#define BOREN_OFF           0xFFBF
//Low-Voltage (Single-Supply) In-Circuit Serial Programming Enable bit
//RB3/PGM pin has PGM function; low-voltage programming enabled
#define LVP_ON              0xFFFF
//RB3 is digital I/O, HV on MCLR must be used for programming
#define LVP_OFF             0xFF7F
//Data EEPROM Memory Code Protection bit
//Data EEPROM code protection off
#define CPD_OFF             0xFFFF
//Data EEPROM code-protected
#define CPD_ON              0xFEFF
//Flash Program Memory Write Enable bits
//Write protection off; all program memory may be written to by EECON control
#define WRT_OFF             0xFFFF
//0000h to 00FFh write-protected; 0100h to 1FFFh may be written to by EECON control
#define WRT_256             0xFDFF
//0000h to 07FFh write-protected; 0800h to 1FFFh may be written to by EECON control
#define WRT_1FOURTH         0xFBFF
//0000h to 0FFFh write-protected; 1000h to 1FFFh may be written to by EECON control
#define WRT_HALF            0xF9FF
//In-Circuit Debugger Mode bit
//In-Circuit Debugger disabled, RB6 and RB7 are general purpose I/O pins
#define DEBUG_OFF           0xFFFF
//In-Circuit Debugger enabled, RB6 and RB7 are dedicated to the debugger
#define DEBUG_ON            0xF7FF
//Flash Program Memory Code Protection bit
//Code protection off
#define CP_OFF              0xFFFF
//All program memory code-protected
#define CP_ON               0xDFFF
```

Przy okazji rozszyfrujemy trzy skróty, bardzo często występujące w dokumentacji:

FOSC (*Frequency of the device OSCillator*) — częstotliwość oscylatora podłączonego do układu. W naszym przykładzie FOSC = 20 MHz.

TOSC (*Time for the device OSCillator*) — czas jednego cyklu zewnętrznego oscylatora. W naszym przykładzie, w którym FOSC = 20 MHz, wielkość TOSC = 1/20000000 s = 0,00000005 s (50 ns).

TCY (*Time for the CYcle*) — czas jednego cyklu maszynowego = 4*TOSC.

Skąd wiemy, że mając w układzie rezonator 20 MHz, powinniśmy wybrać maskę `FOSC_HS`? Niestety, z dokumentacji. Z tabeli, takiej jak 14-2, którą prezentuje rysunek 2.19.

Rysunek 2.19. *Tabela typów oscylatora i zalecanych pojemności kondensatora dla różnych prędkości rezonatora krystalicznego (tabela pochodzi z dokumentacji mikrokontrolera PIC16F877A — DS39582B, s. 146)*

TABLE 14-2: CAPACITOR SELECTION FOR CRYSTAL OSCILLATOR

Osc Type	Crystal Freq.	Cap. Range C1	Cap. Range C2
LP	32 kHz	33 pF	33 pF
	200 kHz	15 pF	15 pF
XT	200 kHz	47-68 pF	47-68 pF
	1 MHz	15 pF	15 pF
	4 MHz	15 pF	15 pF
HS	4 MHz	15 pF	15 pF
	8 MHz	15-33 pF	15-33 pF
	20 MHz	15-33 pF	15-33 pF

Czy z tego wynika, że korzystanie z dokumentacji jest ważne? Cóż, nie jest złe. Ale prawdę mówiąc, komu chciałoby się czytać dokumentację? Chyba że się jest autorem podręcznika. Wtedy raczej wypada[4].

Przechodzimy do definiowania funkcji `main`, czyli do pisania kodu właściwego. Na rysunku 2.5 widzimy, że diodę LED podłączyliśmy do linii RA5. Zainteresujmy się nią, a jest to zdrowe zainteresowanie. Czy przypadkiem linia RA5 nie współdzieli wyprowadzenia z wejściem analogowym? Przypadkiem współdzieli. Jak wynika z rysunku 2.3, wyprowadzenie o numerze 7 może być jednocześnie cyfrową linią portu A, a także — domyślnie — wejściem analogowym AN4. Znów sięgamy do dokumentacji mikrokontrolera PIC16F877A. Chcemy się dowiedzieć, w jaki sposób z AN4 uczynić RA5. Oto zagadka. Pierwszym tropem powinno być znalezienie rejestru `ADCON1` (patrz rysunek 2.20).

Rysunek 2.20. *Organizacja zawartości rejestru ADCON1*

Numer bitu	7	6	5	4	3	2	1	0
Nazwa bitu	ADFM	ADCS2	–	–	PCFG3	PCFG2	PCFG1	PCFG0
RESET	0	0	0	0	0	0	0	0

ADCON1

Ponieważ moduł ADC interesuje nas na razie o tyle, że chcemy go wyłączyć, pomijamy rolę bitów ADFM i ADCS2. Konfigurację kluczowych dla nas bitów PCFG3:PCFG0 znajdziemy w tabeli, którą przedstawia rysunek 2.21.

[4] To oczywiście żart. Od początku tego rozdziału staram się przekonać szanownego Czytelnika, że drobiazgowe i rzetelne przygotowanie najprostszego nawet programu, wraz ze związaną z tym koniecznością sięgania do dokumentacji, przypomina pracę detektywa, a przez to staje się fascynującą przygodą. Oczywiście na początku programistycznej drogi liczba problemów do rozwiązania może zniechęcić najwytrwalszych badaczy, dlatego dobrze jest wtedy korzystać z gotowych i sprawdzonych rozwiązań.

Rysunek 2.21. *Ustawienia bitów PCFG3:PCFG0 i odpowiadająca im konfiguracja linii mikrokontrolera (tabela pochodzi z dokumentacji DS39582B, s. 128)*

PCFG3:PCFG0: A/D Port Configuration Control bits

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O
C/R = # of analog input channels/# of A/D voltage references

Z tabeli odczytujemy następujący fakt: jeśli chcemy, aby wszystkie linie były cyfrowe, konfiguracja bitów PCFG3:PCFG0 powinna odpowiadać liczbie 6 lub 7. W naszym programie (przypominam, że piszemy program) będzie to wyglądało tak.

```
ADCON1 = 0x06;
```

Teraz zaświecimy diodę. Zastanówmy się, dlaczego linie cyfrowe nazywa się liniami I/O, czyli wejścia/wyjścia? Oczywiście wiemy. Jeżeli do linii podłączamy urządzenie, którym chcemy sterować, linia musi być wyjściowa. Czasem jednak chcemy odczytać informację, na przykład pochodzącą z czujnika temperatury. Wtedy linia musi być wejściowa. Tę elementarną wiedzę już mamy. Dodatkowo linia wyjściowa może mieć wysoki lub niski stan logiczny. W mikrokontrolerach PIC rejestrem kierunkowym jest rejestr TRISx, gdzie x oznacza nazwę portu, czyli na przykład TRISA dla portu A, TRISB dla portu B itd. Linię portu konfigurujemy w kierunku wyjściowym, zerując odpowiedni bit rejestru TRISx, natomiast ustawienie bitu czyni linię wejściową. Zapiszmy to w języku C. Załóżmy, że chcemy, by linia RA3 była wejściowa. W tym celu musimy ustawić trzeci bit rejestru TRISA. Możemy to zrobić tak:

```
TRISA |= (1<<3);
```

Można też zastosować zapis binarny:

```
TRISA |= 0b00001000;
```

Można zastosować zapis szesnastkowy:

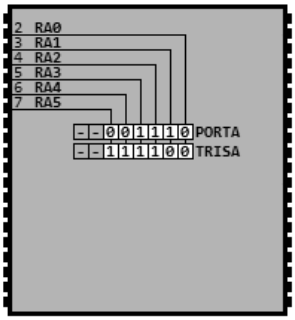
```
TRISA |= 0x08;
```

Zauważmy, że w powyższych przykładach zastosowaliśmy operator sumy bitowej, dzięki czemu pozostałe bity rejestru TRISA pozostały niezmienione. Czasem jednak chcemy od razu skonfigurować wszystkie bity rejestru TRISA. Wtedy stosujemy operator przypisania.

```
TRISA = 0b111100;
PORTA = 0b001110;
```

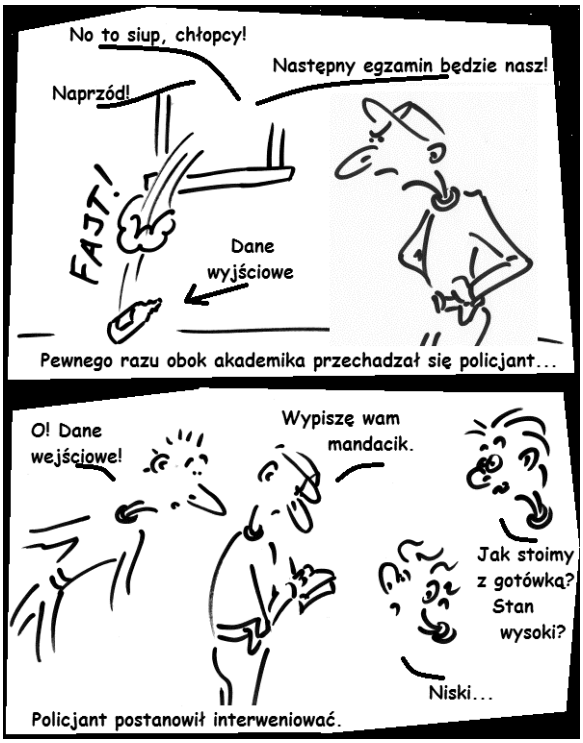
W tym przykładzie tylko dwie linie będą wyjściowe: RA0, która będzie w niskim stanie logicznym, i RA1, której stan będzie wysoki. W sposób obrazowy przedstawia to rysunek 2.22.

Rysunek 2.22. *Schemat pokazujący konfigurowanie linii portu A za pomocą rejestrów PORTA i TRISA*



Aby rzecz była zupełnie jasna, popatrzmy na rysunek 2.23.

Rysunek 2.23. *Opowiastka dydaktyczna pod tytułem „Trudne i pełne wyrzeczeń życie studenta”*



Jeśli nie zależy nam na konkretnej konfiguracji pozostałych linii portu, można po prostu cały rejestr `TRISA` wypełnić zerami i uczynić wszystkie linie wyjściowymi. Tak też postąpimy w naszym programie.

```
TRISA = 0;      //linie portu A wyjściowe
PORTA = (1<<5);      //włącz linię 5.
```

Program mikrokontrolerowy powinien zawsze kończyć się pętlą nieskończoną. To ostatni element naszego kodu.

```
for(;;);          //pętla nieskończona
```

Popatrzmy teraz na cały pierwszy program.

Listing R02_Prog01_C_PIC16F877A.c

```
#include <htc.h>

//oscylator szybszy od 10 MHz (FOSC_HS)
//watchdog wyłączony (WDTE_OFF)
//wyłączone LVP (Low-Voltage ICSP Programming) (LVP_OFF)
__CONFIG(FOSC_HS & WDTE_OFF & LVP_OFF);

void main()
{
    ADCON1 = 0x06;          //wyłączenie linii analogowych
                            //(wszystkie linie cyfrowe)
    TRISA = 0;          //linie portu A wyjściowe
    PORTA = (1<<5);     //włącz linię 5.

    for(;;);          //pętla nieskończona
}
```

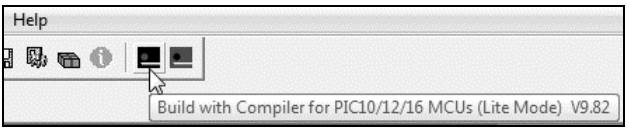
Czy po wgraniu tego programu do mikrokontrolera dioda powinna świecić? Tak, nie ma wyjścia. Ale jako się rzekło, najpierw musimy program skompilować, a następnie wgrać go do układu. Nasz plik z kodem nie jest jeszcze częścią projektu. Aby go dołączyć, klikamy prawym klawiszem myszy w pole *Source Files*, a następnie z rozwiniętej listy wybieramy *Add Files…* (patrz rysunek 2.24).

Rysunek 2.24. *Dołączanie pliku źródłowego do zasobów projektu*

Ponieważ może nam się nieraz zdarzyć — przez roztargnienie — że zapomnimy o dołączeniu pliku źródłowego, przyjrzyjmy się komunikatowi generowanemu w takim przypadku:

```
Error   [939] ; . no file arguments
```

Teraz kompilujemy projekt, naciskając klawisz *F10* lub klikając ikonę *Build with Compiler for PIC10/12/16 MCUs* (patrz rysunek 2.25).



Rysunek 2.25. *Ikona kompilacji projektu*

Jeśli nie było błędów, kompilację zakończy taki komunikat:

```
********** Build successful! **********
```

Dodatkowo, jeśli korzystamy z wersji Lite kompilatora, zostaniemy ostrzeżeni o braku optymalizacji programu.

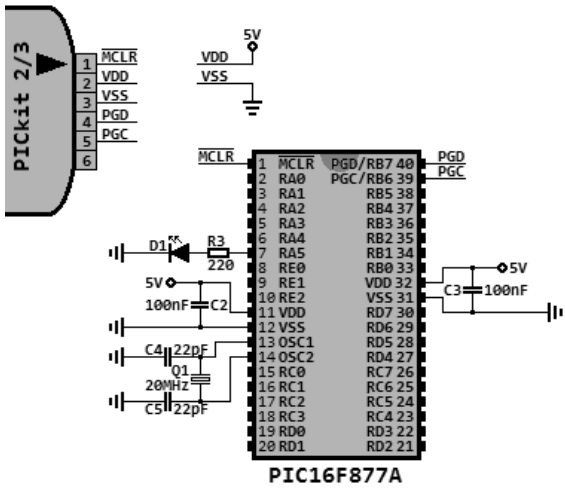
```
(1273) Omniscient Code Generation not available in Lite mode (warning)
```

Jak twierdzi kompilator, gdybyśmy używali płatnej wersji PRO kompilatora, otrzymalibyśmy program wynikowy 40% mniejszy od właśnie utworzonego.

```
Running this compiler in PRO mode, with Omniscient Code Generation enabled,
produces code which is typically 40% smaller than in Lite mode.
See http://microchip.htsoft.com/portal/pic_pro for more information.
```

Nie mam nic przeciwko dodatkowym 40% kodu. Mam nadzieję, że Czytelnik także. Kontynuujmy więc proces programowania.

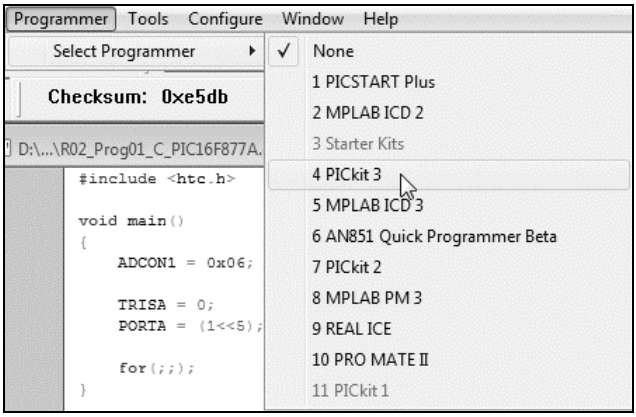
Nadszedł czas, by załadować program do pamięci flash mikrokontrolera. Niezależnie od tego, z jakiego programatora korzystamy, powinniśmy podłączyć go pięcioma liniami do mikrokontrolera. Są to: MCLR, VDD, VSS, PGD i PGC. Czasami konieczne jest odłączenie obwodu RESET na czas programowania układu, szczególnie jeśli linię MCLR bezpośrednio podłączamy do źródła napięcia. Rysunek 2.26 przedstawia schemat podłączenia układu do programatora PICkit 2 bądź PICkit 3.



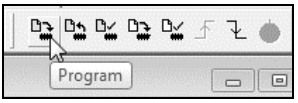
Rysunek 2.26. *Schemat podłączenia układu do programatorów PICkit 2 lub PICkit 3*

Pamiętajmy, by podłączyć zasilanie do układu, a programator podłączyć do komputera. Szczęśliwi posiadacze programatora PICkit 3 mogą w procesie programowania wspomóc się zasobami środowiska MPLAB IDE. Program ładujący uruchamiamy, wybierając z menu typ używanego programatora. W przypadku programatora PICkit 3 jest to ścieżka *Programmer/Select Programmer/PICkit 3* (patrz rysunek 2.27).

Skoro wybór się dokonał, w tym momencie pojawi się okno narzędzi programatora. Wybieramy ikonę *Program* (patrz rysunek 2.28).



Rysunek 2.27. *Wybór programatora*



Rysunek 2.28. *Uruchomienie procesu ładowania programu do pamięci flash mikrokontrolera*

Program został załadowany. Aby układ zaczął działać, do MCLR należy podłączyć obwód RESET, o ile byliśmy zmuszeni go odłączyć. Bardzo możliwe, że będziemy zmuszeni także odłączyć od układu programator. W tym momencie dioda powinna świecić.

Zajmijmy się teraz mruganiem diody. Wciąż programujemy układ z rysunku 2.5. Dużym ułatwieniem będzie dla nas obecność funkcji odmierzających czas. Zostały zaimplementowane w bibliotece kompilatora HI-TECH. Właściwie są to makra postaci:

```
__delay_ms(unsigned long n);     //czekaj n ms
__delay_us(unsigned long n);     //czekaj n µs
```

Korzystanie z nich wymaga wcześniejszego zdefiniowania nazwy _XTAL_FREQ prędkością naszego oscylatora.

```
#define _XTAL_FREQ 20000000
```

Ponieważ program nie zawiera więcej tajemnic, popatrzmy na jego kod.

Listing R02_Prog01_C_PIC16F877A.c

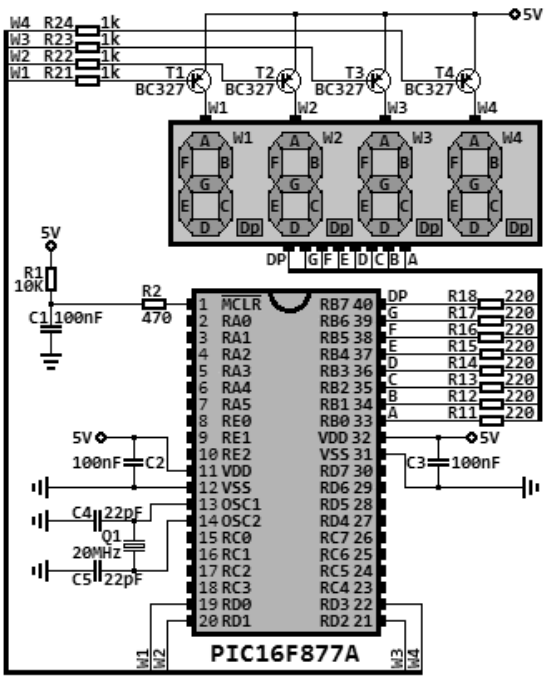
```
#include <htc.h>
//definiujemy szybkość oscylatora dla funkcji __delay_
#define _XTAL_FREQ 20000000

//oscylator szybszy od 10 MHz (FOSC_HS)
//watchdog wyłączony (WDTE_OFF)
//wyłączone LVP (Low-Voltage ICSP Programming) (LVP_OFF)
__CONFIG(FOSC_HS & WDTE_OFF & LVP_OFF);

void main()
{
```

```
    ADCON1 = 0x06;          //wyłączenie linii analogowych
                            //(wszystkie linie cyfrowe)
    TRISA = 0;          //linie portu A wyjściowe

    for(;;)             //pętla nieskończona
    {
        PORTA |= (1<<5);    //włącz linię 5.
        __delay_ms(500);    //czekaj 1/2 s
        PORTA &= ~(1<<5);    //wyłącz linię 5.
        __delay_ms(500);    //czekaj 1/2 s

    }
}
```

Obsługa wyświetlacza LED

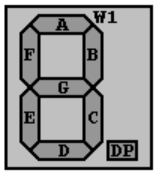
Trzecim programem rozdziału, zgodnie z zapowiedzią, ma być program obsługujący wyświetlacz LED. Schemat będzie nieco bardziej skomplikowany niż ten z rysunku 2.5. Zresztą przekonajmy się sami. Spójrzmy na rysunek 2.29.

Rysunek 2.29. *Schemat układu z mikrokontrolerem PIC16F877A i wyświetlaczem LED*



Wyświetlacz LED to nic innego niż zbiór gustownie ułożonych diod. Pojedynczy wyświetlacz składa się z ośmiu diod, na rysunku 2.30 oznaczonych od A do DP.

Rysunek 2.30.
Ułożenie diod w jednej części wyświetlacza LED



Obsługa wyświetlacza LED jest najłatwiejsza, gdy podłączamy go do mikrokontrolera zgodnie z pewnym porządkiem: diodę A do linii zerowej portu, diodę B do linii pierwszej itd. Jak wynika z rysunku 2.29, diody wyświetlacza będą załączane stanem niskim. Na przykład chcąc wyświetlić cyfrę 3, musimy włączyć (stanem niskim) diody A, B, C, D i G. Pozostałe diody wyłączamy, podając na odpowiadające im linie logiczne 1.

```
PORTB = 0b10110000;      //wyświetl cyfrę 3
```

Jeśli jednak załączane diody są częścią większego wyświetlacza LED, a tak jest w tym przypadku, każdą z części musimy załączyć, podając stan niski na linie tranzystorów T1, T2, T3, T4. Oto dla przykładu bardzo krótki program, którym wyświetlimy w układzie z rysunku 2.29 cyfrę 3.

```
#include <htc.h>

//oscylator szybszy od 10 MHz (FOSC_HS)
//watchdog wyłączony (WDTE_OFF)
//wyłączone LVP (Low-Voltage ICSP Programming) (LVP_OFF)
__CONFIG(FOSC_HS & WDTE_OFF & LVP_OFF);

void main()
{
    ADCON1 = 0x06;              //wyłączenie linii analogowych
                                //(wszystkie linie cyfrowe)
    TRISB = 0;                  //wszystkie linie portu B wyjściowe
    TRISD = 0;                  //wszystkie linie portu D wyjściowe
    PORTD = 0b00001110;         //włącz W1 (RD0 w stan logicznego 0)
                                //W2, W3, W4 wyłączone
    PORTB = 0b10110000;         //wyświetl cyfrę 3
    for(;;);                    //pętla nieskończona
}
```

Teraz nasza ambicja każe nam jednocześnie obsłużyć cztery części wyświetlacza, aby wyświetlić na nim liczbę 1234. Nie jest to zadanie nie do zrealizowania. Wystarczy zrozumieć mechanizm załączania wyświetlaczy. Otóż chcąc wyświetlić jednocześnie cztery różne cyfry, należy każdą część wyświetlacza na chwilę włączyć i wyświetlić na niej pożądaną cyfrę. Czas załączenia części wyświetlacza nie powinien być za długi (cyfry będą migać) ani za krótki (cyfry będą słabo widoczne). W naszym kolejnym programie czas załączania będzie wynosił 5 ms.

Podłączamy wyświetlacz LED zgodnie ze schematem z rysunku 2.29. Najłatwiej będzie nam to zrobić na płytce edukacyjnej ARE. Linie portu B mikrokontrolera podłączamy do kolejnych pinów przy wyświetlaczu LED. I tak linię RB0 podłączamy do pinu A, RB1 do B itd. Cztery linie portu D podłączamy do pinów załączających tranzystory: linię RD0 do W1, RD1 do W2 itd. Mając zbudowany układ elektroniczny, wypróbujmy taki program.

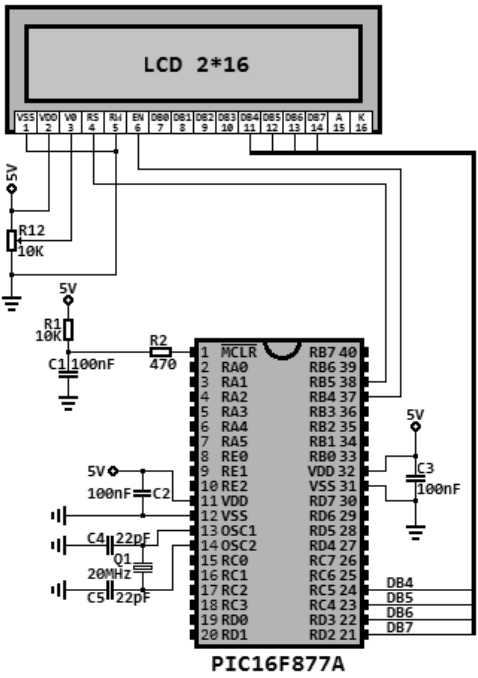
Listing R02_Prog03_C_PIC16F877A.c

```
//definiujemy szybkość oscylatora dla funkcji __delay_
#define _XTAL_FREQ 20000000
#include <htc.h>

//oscylator szybszy od 10 MHz (FOSC_HS)
//watchdog wyłączony (WDTE_OFF)
//wyłączone LVP (Low-Voltage ICSP Programming) (LVP_OFF)
__CONFIG(FOSC_HS & WDTE_OFF & LVP_OFF);

void main()
{
    ADCON1 = 0x06;              //wyłączenie linii analogowych
                                    //(wszystkie linie cyfrowe)
    TRISB = 0;              //wszystkie linie portu B wyjściowe
    TRISD = 0;              //wszystkie linie portu D wyjściowe

    for(;;)                 //pętla nieskończona
    {
        PORTD = 0b00001110;     //włącz W1 (RD0 w stan logicznego 0)
                    //W2, W3, W4 wyłączone
        PORTB = 0b11111001;     //wyświetl cyfrę 1
        __delay_ms(5);          //zaczekaj 5 ms

        PORTD = 0b00001101;     //włącz W2 (RD1 w stan logicznego 0)
                                //W1, W3, W4 wyłączone
        PORTB = 0b10100100;     //wyświetl cyfrę 2
        __delay_ms(5);          //zaczekaj 5 ms

        PORTD = 0b00001011;     //włącz W3 (RD2 w stan logicznego 0)
                    //W1, W2, W4 wyłączone
        PORTB = 0b10110000;     //wyświetl cyfrę 3
        __delay_ms(5);          //zaczekaj 5 ms

        PORTD = 0b00000111;     //włącz W4 (RD3 w stan logicznego 0)
                        //W1, W2, W3 wyłączone
        PORTB = 0b10011001;     //wyświetl cyfrę 4
        __delay_ms(5);          //zaczekaj 5 ms
    }
}
```

Obsługa wyświetlacza alfanumerycznego LCD

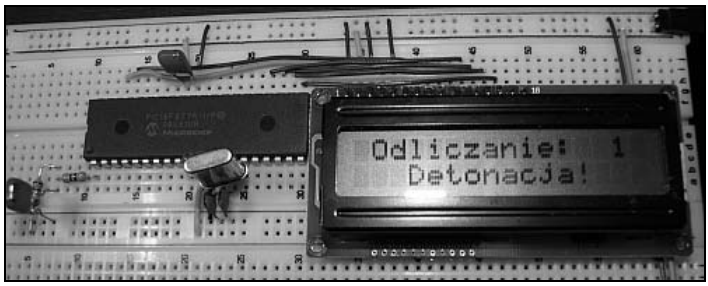
Więcej z diod wycisnąć się nie da. Można na wyświetlaczu LED generować liczby zmieniające się dynamicznie, na przykład zgodnie z upływającym czasem. Zostawiam ten temat radosnej twórczości Czytelnika. Nadszedł czas, by zapoznać się z tematem obsługi wyświetlacza alfanumerycznego LCD. Schemat układu, który będziemy obsługiwać, został przedstawiony na rysunku 2.31.

Kompilator HI-TECH ma własną bibliotekę obsługi wyświetlaczy LCD opartych na standardowym sterowniku Hitachi HD44780. Niestety, owa biblioteka ma wiele ograniczeń. Na przykład działa tylko dla jednego układu podłączeń. Tymczasem często jest tak,



Rysunek 2.31. *Schemat układu z mikrokontrolerem PIC16F877A i alfanumerycznym wyświetlaczem LCD*

że podłączamy wyświetlacz po prostu do linii, które nie są zajęte. Proponuję zatem, abyśmy zbudowali własną bibliotekę, wolną od wymienionych niedoskonałości. Zanim jednak do tego dojdzie, zbudujmy próbny układ. Rysunek 2.32 przedstawia realizację układu z rysunku 2.31 na płytce stykowej.

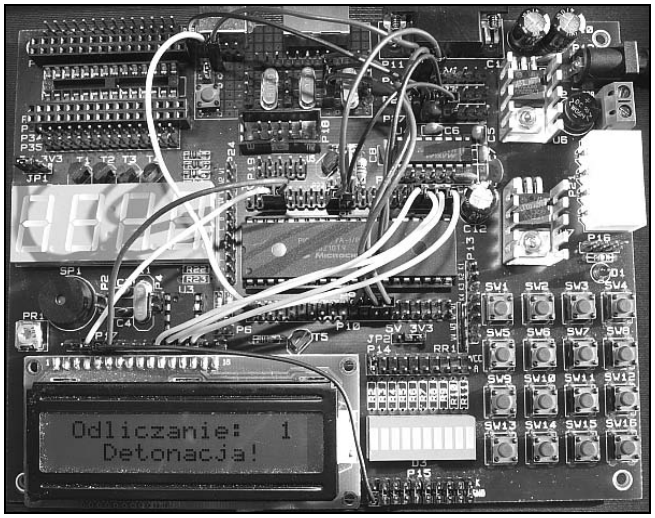


Rysunek 2.32. *Realizacja układu z mikrokontrolerem PIC16F877A i alfanumerycznym wyświetlaczem LCD na płytce stykowej*

Na rysunku 2.33 widzimy ten sam układ zrealizowany na płytce edukacyjnej.

Realizując układ na płytce edukacyjnej ARE, należy linię RW wyświetlacza LCD samodzielnie podłączyć do masy.

Zbudowana przez nas biblioteka obsługi wyświetlacza LCD ma poprawnie działać dla każdej konfiguracji podłączenia wyprowadzeń mikrokontrolera. W tym kontekście najłatwiejsze do zaprogramowania byłyby dla nas rozkazy, w których będziemy mogli użyć nazw pojedynczych bitów. Na szczęście istnieje taka możliwość — dzięki zdefiniowaniu specjalnych struktur. Oto fragment pochodzący z pliku *pic16f877a.h* znajdującego się w zasobach kompilatora HI-TECH.



Rysunek 2.33. *Realizacja układu z mikrokontrolerem PIC16F877A i alfanumerycznym wyświetlaczem LCD na płytce edukacyjnej*

Listing pic16f877a.h (fragment)

```
//Register: PORTB
volatile unsigned char          PORTB               @ 0x006;
//bit and bitfield definitions
volatile bit RB0                @ ((unsigned)&PORTB*8)+0;
volatile bit RB1                @ ((unsigned)&PORTB*8)+1;
volatile bit RB2                @ ((unsigned)&PORTB*8)+2;
volatile bit RB3                @ ((unsigned)&PORTB*8)+3;
volatile bit RB4                @ ((unsigned)&PORTB*8)+4;
volatile bit RB5                @ ((unsigned)&PORTB*8)+5;
volatile bit RB6                @ ((unsigned)&PORTB*8)+6;
volatile bit RB7                @ ((unsigned)&PORTB*8)+7;
#ifndef _LIB_BUILD
volatile union {
    struct {
        unsigned    RB0             : 1;
        unsigned    RB1             : 1;
        unsigned    RB2             : 1;
        unsigned    RB3             : 1;
        unsigned    RB4             : 1;
        unsigned    RB5             : 1;
        unsigned    RB6             : 1;
        unsigned    RB7             : 1;
    };
} PORTBbits @ 0x006;
#endif

//Register: TRISB
volatile unsigned char          TRISB               @ 0x086;
//bit and bitfield definitions
volatile bit TRISB0             @ ((unsigned)&TRISB*8)+0;
volatile bit TRISB1             @ ((unsigned)&TRISB*8)+1;
volatile bit TRISB2             @ ((unsigned)&TRISB*8)+2;
volatile bit TRISB3             @ ((unsigned)&TRISB*8)+3;
volatile bit TRISB4             @ ((unsigned)&TRISB*8)+4;
volatile bit TRISB5             @ ((unsigned)&TRISB*8)+5;
```

```
volatile bit TRISB6             @ ((unsigned)&TRISB*8)+6;
volatile bit TRISB7             @ ((unsigned)&TRISB*8)+7;
#ifndef _LIB_BUILD
volatile union {
    struct {
        unsigned    TRISB0              : 1;
        unsigned    TRISB1              : 1;
        unsigned    TRISB2              : 1;
        unsigned    TRISB3              : 1;
        unsigned    TRISB4              : 1;
        unsigned    TRISB5              : 1;
        unsigned    TRISB6              : 1;
        unsigned    TRISB7              : 1;
    };
} TRISBbits @ 0x086;
#endif
```

Jak widać, wybrany fragment dotyczy portu B. Co z niego wynika? Otóż dzięki tym definicjom możliwe jest użycie w kodzie takiego zapisu:

```
TRISBbits.TRISB5 = 0;
PORTBbits.RB5 = 1;
```

Co więcej, nazwy bitów zastąpimy nazwami linii wyświetlacza. To pozwoli uczynić naszą bibliotekę obsługi wyświetlacza naprawdę uniwersalną. Przy tym umawiamy się, że przed użyciem naszej biblioteki zawsze konieczne będzie zdefiniowanie tych nazw. W przypadku układu z rysunku 2.31 definicja nazw linii wyświetlacza powinna wyglądać tak:

```
#define TRIS_RS_LCD   TRISBbits.TRISB5
#define TRIS_EN_LCD   TRISBbits.TRISB4
#define TRIS_DB4_LCD  TRISCbits.TRISC5
#define TRIS_DB5_LCD  TRISCbits.TRISC4
#define TRIS_DB6_LCD  TRISDbits.TRISD3
#define TRIS_DB7_LCD  TRISDbits.TRISD2

#define RS_LCD   PORTBbits.RB5
#define EN_LCD   PORTBbits.RB4
#define DB4_LCD  PORTCbits.RC5
#define DB5_LCD  PORTCbits.RC4
#define DB6_LCD  PORTDbits.RD3
#define DB7_LCD  PORTDbits.RD2

#include "lcd.h"
```

Jeszcze dwa słowa o rolach linii RS, EN, DB4-DB7. Co do ostatnich czterech mamy podejrzenie, że służą przesyłaniu danych. Tak jest w istocie. Linia EN sygnalizuje ważny transfer danych. A dokładniej: potwierdzamy transfer ważnych danych opadającym zboczem EN[5]. Linią RS sygnalizujemy przesyłanie rozkazu (stan niski) lub danych (stan wysoki).

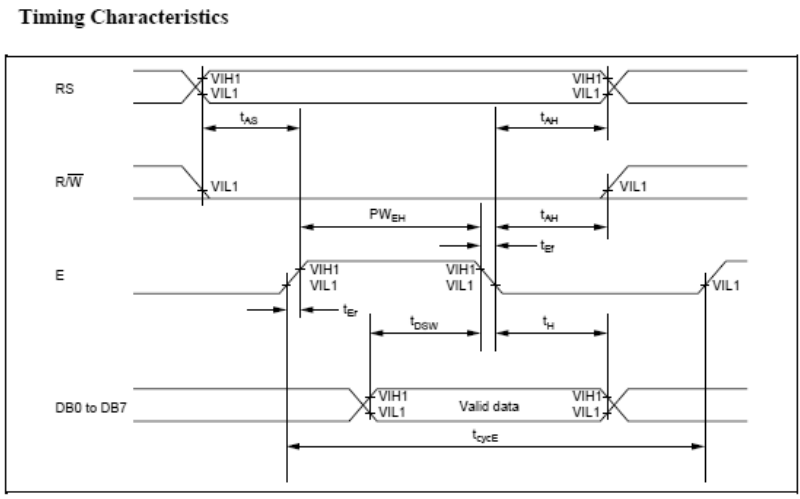
[5] Czytelnikom nieznającym pojęcia zbocza opadającego/narastającego polecam podręcznik *AVR&ARM7. Programowanie mikrokontrolerów dla każdego*, w którym na stronie 236 rzecz wytłumaczyłem dokładniej. A kto ślubował, że przeczyta w swoim życiu tylko tę jedną książkę, którą właśnie czyta, tego proszę o cierpliwość. Do zagadnienia zbocza opadającego/narastającego wrócę jeszcze w rozdziale 4 przy okazji tematu przerwań.

Zdefiniujemy funkcję wysyłającą bajt do pamięci sterownika wyświetlacza LCD. Ponieważ zdecydowaliśmy się na interfejs 4-bitowy, musimy wysyłać dane w dwóch paczkach po 4 bity. W pierwszej kolejności wyślemy 4 najstarsze bity.

```
//ustawienie EN
  EN_LCD = 1;
//wysłanie 4 najstarszych bitów danych
  if(bajt & 0x80)    DB7_LCD    = 1; else DB7_LCD = 0;
  if(bajt & 0x40)    DB6_LCD    = 1; else DB6_LCD = 0;
  if(bajt & 0x20)    DB5_LCD    = 1; else DB5_LCD = 0;
  if(bajt & 0x10)    DB4_LCD    = 1; else DB4_LCD = 0;
```

Linię EN ustawiamy, żeby móc ją następnie wyzerować. Przecież to właśnie opadające zbocze tej linii ma potwierdzić ważność danych. Tu powstaje pytanie: po jakim czasie można wysłać następną paczkę danych? Sprawdźmy w dokumentacji sterownika HD44780 (patrz rysunek 2.34).

Rysunek 2.34. *Przebiegi czasowe dla operacji zapisywania danych (rysunek pochodzi z dokumentacji sterownika HD44780, s. 58)*



Wartości liczbowe odcinków czasu zaznaczonych na rysunku 2.34 znajdziemy w tabeli, którą przedstawia rysunek 2.35.

Rysunek 2.35. *Wartości liczbowe przebiegów czasowych dla operacji zapisywania danych (rysunek pochodzi z dokumentacji sterownika HD44780, s. 52)*

Bus Timing Characteristics

Write Operation

Item	Symbol	Min	Typ	Max	Unit
Enable cycle time	t_{cycE}	500	—	—	ns
Enable pulse width (high level)	PW_{EH}	230	—	—	
Enable rise/fall time	t_{Er}, t_{Ef}	—	—	20	
Address set-up time (RS, R/$\overline{W}$ to E)	t_{AS}	40	—	—	
Address hold time	t_{AH}	10	—	—	
Data set-up time	t_{DSW}	80	—	—	
Data hold time	t_{H}	10	—	—	

Jeden pełny cykl linii EN, oznaczony na rysunku 2.34 jako t_{cycE}, nie może być krótszy niż 500 ns, czyli 0,5 µs. Natomiast minimalna długość stanu wysokiego na linii EN, na rysunku 2.34 oznaczona jako PW_{EH}, nie może być krótsza od 230 ns. Należy o tym pamiętać, szczególnie przy programowaniu układów o szybkim taktowaniu. Dlatego między kolejnymi transferami paczki danych umieścimy funkcję oczekującą 1 µs. Dodatkowo damy opóźnienie przed każdym potwierdzaniem ważnych danych. Natomiast całą operację zapisywania danych powinien zakończyć czas oczekiwania 37 µs. Skąd to wiem? Z tabeli, którą właśnie mam przyjemność przedstawić (tabela 2.1).

Skoro operacja zapisywania danych ma trwać nie więcej niż 40 µs, a w funkcji umieszczamy już trzy funkcje opóźniające o 1 µs, należy dodać opóźnienie 37 µs. A oto cała funkcja wysyłająca dane do sterownika wyświetlacza LCD.

Listing lcd.h (fragment)

```
void Wyslij_do_LCD(unsigned char bajt)
{
  //ustawienie EN
    EN_LCD = 1;
  //wysłanie 4 najstarszych bitów danych
    if(bajt & 0x80)    DB7_LCD    = 1; else DB7_LCD = 0;
    if(bajt & 0x40)    DB6_LCD    = 1; else DB6_LCD = 0;
    if(bajt & 0x20)    DB5_LCD    = 1; else DB5_LCD = 0;
    if(bajt & 0x10)    DB4_LCD    = 1; else DB4_LCD = 0;
  //zaczekaj 1 µs
    __delay_us(1);
  //potwierdzenie wysłania danych (opadającym zboczem EN)
    EN_LCD = 0;

  //zaczekaj 1 µs
    __delay_us(1);

  //ustawienie EN
    EN_LCD = 1;
  //wysłanie 4 najmłodszych bitów danych
   if(bajt & 0x08)   DB7_LCD   = 1; else DB7_LCD = 0;
   if(bajt & 0x04)   DB6_LCD   = 1; else DB6_LCD = 0;
   if(bajt & 0x02)   DB5_LCD   = 1; else DB5_LCD = 0;
   if(bajt & 0x01)   DB4_LCD   = 1; else DB4_LCD = 0;
  //zaczekaj 1 µs
    __delay_us(1);
  //potwierdzenie wysłania danych (opadającym zboczem EN)
    EN_LCD = 0;

  //zaczekaj 37 µs
    __delay_us(37);
}
```

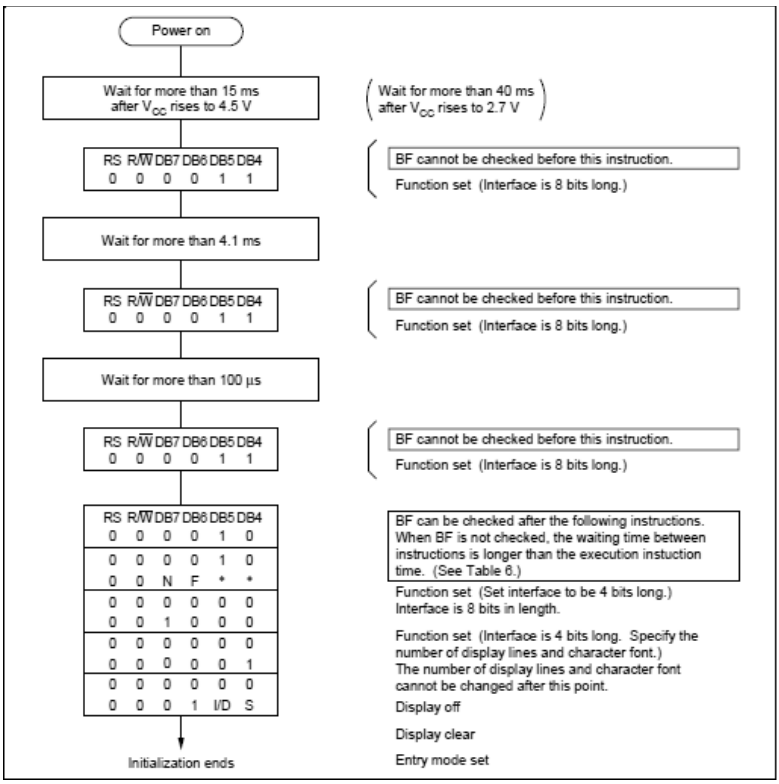
Świat jest tak przedziwnie zbudowany, że aby samochód mógł jechać, najpierw musi ruszyć. Ta zdumiewająca własność dotyczy także alfanumerycznych wyświetlaczy LCD. Aby móc na nich cokolwiek wyświetlić, w pierwszej kolejności trzeba je inicjalizować. Algorytm tej czynności znajdziemy w dokumentacji sterownika. Oczywiście nas interesuje inicjalizacja interfejsu 4-bitowego, którą prezentuje rysunek 2.36.

Tabela 2.1. *Zestawienie instrukcji sterownika HD44780*

Instrukcja	Kod										Opis	Maksymalny czas wykonania
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Czyść wyświetlacz	0	0	0	0	0	0	0	0	0	1	Czyszczenie zawartości pamięci wyświetlacza, kursor wraca do pozycji początkowej	1,64 ms
Wróć kursor	0	0	0	0	0	0	0	0	1	-	Przywrócenie kursora do pozycji początkowej oraz wyświetlania danych od adresu 0 DDRAM	1,64 ms
Wprowadzanie danych	0	0	0	0	0	0	0	1	I/D	S	I/D — adres po zapisie danych (0: dekrementacja, 1: inkrementacja), S — przesuń po zapisie (0: kursor, 1: okno)	40 µs
Włącz funkcje wyświetlacza	0	0	0	0	0	0	1	D	C	B	D — ustaw wyświetlacz (1: włączony), C — ustaw kursor (1: włączony), B — ustaw miganie kursora (1: włączone)	40 µs
Przesuń okno lub kursor	0	0	0	0	0	1	S/C	R/L	-	-	Przesuń S/C = 0: kursor, S/C = 1: okno, o krok w R/L = 0: lewo, R/L = 1: prawo	40 µs
Ustawienia wyświetlacza	0	0	0	0	1	DL	N	F	-	-	Konfiguruj ustawienia wyświetlacza: DL — liczba linii danych (0: transfer czterema liniami, 1: transfer ośmioma liniami), N — liczba linii wyświetlacza (0: jedna linia, 1: dwie linie), F — rozmiar matrycy znakowej (0: 5×8 punktów, 1: 5×10 punktów)	40 µs

Tabela 2.1. *Zestawienie instrukcji sterownika HD44780 — ciąg dalszy*

Instrukcja	Kod										Opis	Maksymalny czas wykonania
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Ustaw CGRAM	0	0	0	1	A	A	A	L	L	L	Ustaw jako aktywną pamięć CGRAM, ustaw jej nowy 3-bitowy adres (AAA) i numer linii wstawiania (LLL)	40 µs
Ustaw DDRAM	0	0	1	A	A	A	A	A	A	A	Ustaw jako aktywną pamięć DDRAM, ustaw jej nowy 7-bitowy adres (AAAAAAA)	40 µs
Odczytaj flagę zajętości i pamięć	0	1	BF	A	A	A	A	A	A	A	Odczytaj flagę zajętości (BF) i 7-bitowy adres danych (AAAAAAA)	0 µs
Zapisz dane do CGRAM lub DDRAM	1	0	Dane do zapisu								Zapisanie danych do pamięci CGRAM (jeśli wcześniej była komenda Ustaw CGRAM) lub DDRAM	40 µs
Czytaj dane z CGRAM lub DDRAM	1	1	Odczytane dane								Odczytanie danych z pamięci CGRAM (jeśli wcześniej była komenda Ustaw CGRAM) lub DDRAM	40 µs



Rysunek 2.36. *Algorytm inicjalizacji sterownika HD44780 w przypadku interfejsu 4-bitowego (rysunek pochodzi z dokumentacji sterownika, s. 46)*

Szanowny Czytelnik zapewne się domyślił, że skoro przedstawiłem pewien algorytm, to zaraz powiem, dlaczego mi się nie podoba. Tak jest w istocie. Moje doświadczenie pokazuje, że dokładne jego przestrzeganie nie zawsze skutkuje poprawnie zainicjowanym sterownikiem HD44780. Zmienimy dwie rzeczy: wydłużymy czas oczekiwania na stabilizację do 45 ms oraz dodamy czas oczekiwania 100 µs także po trzeciej procedurze wysyłania sekwencji 0011. Oto nasz poprawiony algorytm — co prawda przedstawiłem go już w podręczniku *AVR&ARM7*, ale zrobiłem to z takim wdziękiem, że nie mogę się powstrzymać, by uczynić to jeszcze raz.

1. Ustaw linie `RS_LCD`, `EN_LCD`, `DB4_LCD`, `DB5_LCD`, `DB6_LCD`, `DB7_LCD` w kierunku wyjściowym, wyzeruj linie.
2. Zaczekaj co najmniej 45 ms na ustabilizowanie napięcia.
3. Wyślij sekwencję 0011.
4. Zaczekaj co najmniej 4,1 ms.
5. Powtórnie wyślij sekwencję 0011.
6. Zaczekaj co najmniej 100 µs.
7. Po raz trzeci wyślij sekwencję 0011.
8. Zaczekaj co najmniej 100 µs.

9. Ustaw interfejs 4-bitowy, czyli wyślij sekwencję 0010.
10. Ustaw parametry wyświetlacza.
11. Ustaw tryb pracy wyświetlacza.
12. Włącz wyświetlacz.
13. Wyczyść pamięć wyświetlacza.

A oto programowa realizacja algorytmu.

Listing lcd.h (fragment)

```
void WlaczLCD()
{
  //ustawienie kierunku wyjściowego linii podłączonych do LCD
    TRIS_RS_LCD = 0;
    TRIS_EN_LCD = 0;
    TRIS_DB7_LCD = 0;
    TRIS_DB6_LCD = 0;
    TRIS_DB5_LCD = 0;
    TRIS_DB4_LCD = 0;

  //stan niski na liniach
    RS_LCD = 0;
    EN_LCD = 0;
    DB7_LCD = 0;
    DB6_LCD = 0;
    DB5_LCD = 0;
    DB4_LCD = 0;

  //zaczekaj co najmniej 45 ms na ustabilizowanie się napięcia
    __delay_us(45000);

  //1
  //ustaw linię EN
    EN_LCD = 1;
  //załaduj sekwencję 0011
    DB7_LCD = 0;
    DB6_LCD = 0;
    DB5_LCD = 1;
    DB4_LCD = 1;
  //zaczekaj 1 µs
    __delay_us(1);
  //potwierdź opadającym zboczem EN
    EN_LCD = 0;

  //zaczekaj co najmniej 4,1 ms
    __delay_us(4100);

  //2
  //ustaw linię EN
    EN_LCD = 1;
  //zaczekaj 1 µs
    __delay_us(1);
  //potwierdź opadającym zboczem EN sekwencję 0011
    EN_LCD = 0;
```

```
  //zaczekaj co najmniej 100 µs
    __delay_us(100);

  //3
  //ustaw linię EN
    EN_LCD = 1;
  //zaczekaj 1 µs
    __delay_us(1);
  //potwierdź opadającym zboczem EN sekwencję 0011
    EN_LCD = 0;

  //zaczekaj co najmniej 100 µs
    __delay_us(100);

  //4
  //ustaw linię EN
    EN_LCD = 1;
  //ustawienie interfejsu 4-bitowego
    DB4_LCD = 0;
  //zaczekaj 1 µs
    __delay_us(1);
  //potwierdź opadającym zboczem EN
    EN_LCD = 0;

  //ustaw parametry wyświetlacza
  //bit 4 = 0 (słowo danych ma 4 bity)
  //bit 3 = 1 (2 wiersze znaków)
  //bit 2 = 0 (matryca 5×8 pikseli)
    RS_LCD = 0;
    Wyslij_do_LCD(0b00101000);
    RS_LCD = 1;

  //włącz wyświetlacz
  //bit 2 = 1 (włączenie wyświetlania)
  //bit 1 = 0 (kursor nieaktywny)
  //bit 0 = 0 (kursor nie miga)
    RS_LCD = 0;
    Wyslij_do_LCD(0b00001100);
    RS_LCD = 1;

  //ustaw tryb pracy wyświetlacza
  //bit 2 = 1 (inkrementowanie adresu zapisu danych)
  //bit 1 = 1 (wyłączenie przesuwania w prawo)
    RS_LCD = 0;
    Wyslij_do_LCD(0b00000110);
    RS_LCD = 1;

    CzyscLCD();
}
```

W przedstawionym kodzie występuje wywołanie funkcji `CzyscLCD`, której jeszcze nie zdefiniowaliśmy. Drobne przeoczenie.

Listing lcd.h (fragment)

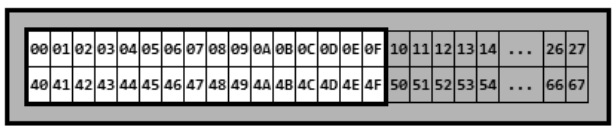
```
void CzyscLCD()
{
   RS_LCD = 0;
   Wyslij_do_LCD(1);
   RS_LCD = 1;

  //czekaj 1.64 ms
   __delay_us(1640);
}
```

Standardowy zestaw znaków sterownika HD44780 pokrywa się w swojej podstawowej części ze znakami kodowania ASCII. To upraszcza zadanie wyświetlania danych na wyświetlaczu. Zdefiniujemy funkcję, której argumentami będą ciąg znaków do wyświetlania wraz z długością napisu.

Listing lcd.h (fragment)

```
void WyswietlLCD(char *napis, unsigned char ile)
{
   unsigned char k = 0;
   while(k<ile)
   {
      Wyslij_do_LCD(napis[k]);
      k++;
   }
}
```

Wróćmy na chwilę do tabeli 2.1. Znajduje się w niej instrukcja Ustaw DDRAM. Dzięki niej możliwy jest zapis znaku w dowolnym miejscu pamięci DDRAM i — co z tego wynika — w dowolnym miejscu ekranu wyświetlacza LCD. Jest to dla nas okoliczność wielce sprzyjająca, chcemy bowiem zbudować funkcję, która pozwoli nam wyświetlać dane w różnych miejscach ekranu. W tym miejscu wypada nam zaznajomić się z architekturą pamięci DDRAM wyświetlacza. Sterownik HD44780 obsługuje 80 komórek pamięci DDRAM, z których każda ma rozmiar 1 bajta. W przypadku wyświetlaczy o dwóch liniach każda z linii ma 40 komórek o adresach podanych na rysunku 2.37.

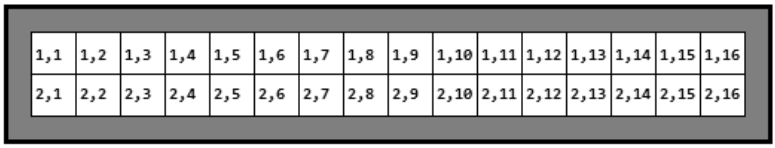


Rysunek 2.37. *Sposób adresowania komórek wyświetlacza LCD typu 2*16*

Zwracam uwagę, że adresy zostały zapisane w kodzie szesnastkowym. Pierwszych 40 komórek ma adresy od 0x00 do 0x27. Adresy komórek w wierszu drugim zaczynają się od liczby 0x40, a kończą liczbą 0x67. Dodatkowo na rysunku zostały zaznaczone widoczne komórki pamięci. Możliwe jest przesuwanie okna wyświetlacza w celu wizualizacji pozostałych adresów pamięci DDRAM.

Po zresetowaniu pamięci wyświetlacza kursor pamięci DDRAM jest umiejscawiany pod adresem 0x00. Zapisanie komórki powoduje automatyczne przesunięcie wskaźnika pod adres następny. Z opisu instrukcji Ustaw DDRAM wiemy, że w celu przesunięcia wskaźnika pod dany adres należy wysłać bajt postaci 1AAAAAAA, składający się z 7-bitowego adresu AAAAAAA oraz ustawionego najstarszego bitu. Wysłanie rozkazu powinno być poprzedzone wyzerowaniem linii RS. A co z listą argumentów budowanej funkcji? Rysunek 2.38 przedstawia zwyczajowe numerowanie komórek ekranu wyświetlacza typu 2*16.

Rysunek 2.38. *Zwyczajowe numerowanie komórek ekranu wyświetlacza LCD typu 2*16*



Z rysunku 2.38 wynika na przykład, że ustawienie kursora w miejscu o współrzędnych (2,14) powinno skutkować przesunięciem wskaźnika pod adres 0x4D. Najprostsza funkcja realizująca to zadanie może wyglądać tak.

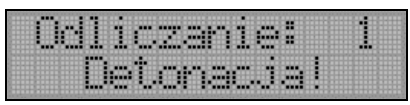
Listing lcd.h (fragment)

```
void UstawKursorLCD(unsigned char y, unsigned char x)
{
  //ustal nowy adres pamięci DD RAM
    unsigned char n;
    if (y==1) n = x - 1;
    else n = 0x40 + x - 1;
  //ustaw kod
    n |= 0b10000000;

  //wyślij rozkaz ustawienia nowego adresu DD RAM
    RS_LCD = 0;
    Wyslij_do_LCD(n);
    RS_LCD = 1;
}
```

Z tabeli 2.1 wyczytamy, że instrukcja ustawienia adresu pamięci DDRAM powinna trwać 40 µs. Nie ma potrzeby dodawania instrukcji opóźniającej, gdyż mniej więcej 40 µs trwa wykonanie funkcji Wyslij_do_LCD.

Zbudowaliśmy własną bibliotekę obsługi alfanumerycznego wyświetlacza LCD. Skorzystamy z niej w sposób właściwy. W pierwszej linii ekranu wyświetlimy napis „Odliczanie:" wraz ze zmieniającymi się co sekundę cyframi od 9 do 1. Po wyświetleniu cyfry 1 w drugiej linii ekranu wyświetlony zostanie napis „Detonacja!". Taki mały żart. Wszystko powinno wyglądać tak jak na rysunku 2.39.

Rysunek 2.39. *Ekran wyświetlacza po wykonaniu bombowego zadania*



Pamiętamy, że skorzystanie z naszej biblioteki obsługi wyświetlacza LCD wymaga zdefiniowania połączenia wyświetlacza do mikrokontrolera. Dopiero po tym zabiegu możemy dołączyć zasoby biblioteki do programu głównego. Drugą rzeczą wymagającą wyjaśnienia jest wyświetlanie liczb. Ponieważ możemy wyświetlać jedynie napisy, skorzystamy z funkcji `utoa` znajdującej się w zasobach kompilatora HI-TECH. Funkcja konwertuje liczby bezznakowe w napisy. W zasobach języka C kompilatora HI-TECH znajdziemy cztery funkcje konwertujące liczby w napisy (patrz tabela 2.2).

Tabela 2.2. *Funkcje konwertujące liczby w napisy*

Deklaracja funkcji	Opis	Przykład użycia

`char *ftoa(float f, int *status);`	Funkcja konwertująca liczbę zmiennoprzecinkową `f` w napis, który jest umieszczany w pamięci o adresie `*buf`.	`char *buf;`
		`float input = 12.34;`
		`int status;`
		`buf = ftoa(input, &status);`

`char *itoa(char *buf, int val, int base)`	Funkcja konwertująca liczbę całkowitą `val` w napis, który jest umieszczany w pamięci o adresie `*buf`. Parametr `base` oznacza podstawę kodowania liczby.	`char buf[10];` `itoa(buf, 1234, 16);`
`char *ltoa(char *buf, long val, int base)`	Funkcja konwertująca liczbę `val` typu `long` w napis, który jest umieszczany w pamięci o adresie `*buf`. Parametr `base` oznacza podstawę kodowania liczby.	`char buf[10];` `utoi(buf, 12345678L, 16);`
`char *utoa(char *buf, unsigned val, int base)`	Funkcja konwertująca liczbę bezznakową `val` w napis, który jest umieszczany w pamięci o adresie `*buf`. Parametr `base` oznacza podstawę kodowania liczby.	`char buf[10];` `utoi(buf, 1234, 16);`

Użycie wymienionych funkcji wymaga dołączenia do programu biblioteki `stdlib`. Zrealizowane zadanie zostało przedstawione na listingu *R02_Prog04_C_PIC16F877A.c*.

Listing R02_Prog04_C_PIC16F877A.c

```
//definiujemy szybkość oscylatora dla funkcji __delay_
#define _XTAL_FREQ 20000000
#include <htc.h>
#include <stdlib.h>

#define TRIS_RS_LCD   TRISBbits.TRISB5
#define TRIS_EN_LCD   TRISBbits.TRISB4
#define TRIS_DB4_LCD  TRISCbits.TRISC5
#define TRIS_DB5_LCD  TRISCbits.TRISC4
#define TRIS_DB6_LCD  TRISDbits.TRISD3
#define TRIS_DB7_LCD  TRISDbits.TRISD2

#define RS_LCD   PORTBbits.RB5
#define EN_LCD   PORTBbits.RB4
#define DB4_LCD  PORTCbits.RC5
#define DB5_LCD  PORTCbits.RC4
#define DB6_LCD  PORTDbits.RD3
#define DB7_LCD  PORTDbits.RD2

#include "lcd.h"
```

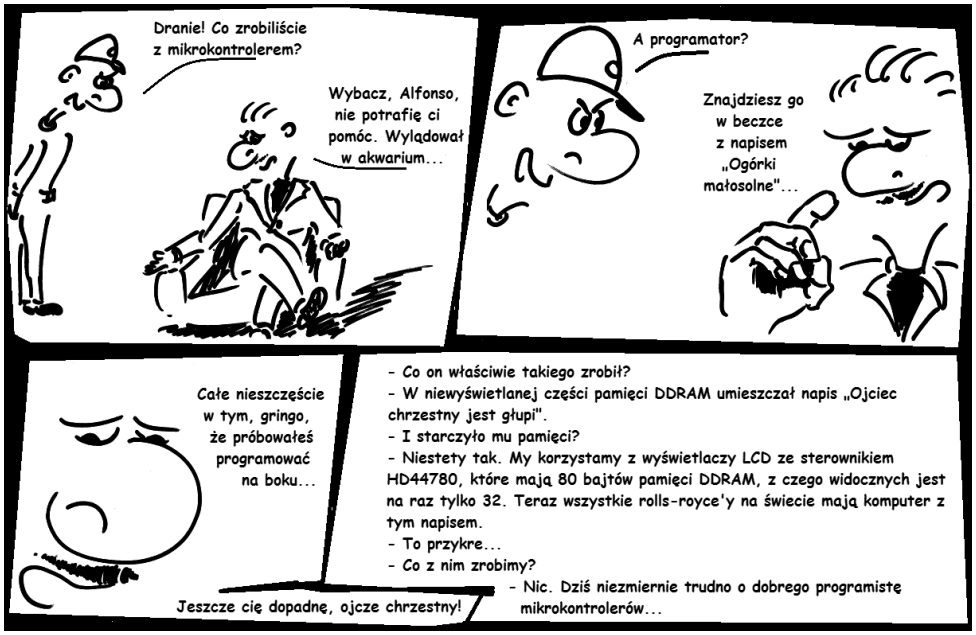
```
//oscylator szybszy od 10 MHz (FOSC_HS)
//watchdog wyłączony (WDTE_OFF)
//wyłączone LVP (Low-Voltage ICSP Programming) (LVP_OFF)
__CONFIG(FOSC_HS & WDTE_OFF & LVP_OFF);

char napis1[] = "Odliczanie:";
char napis2[] = "Detonacja!";
char bufor[2];

void main()
{
   unsigned char i;
   ADCON1 = 0x06;          //wyłączenie linii analogowych
                  ///(wszystkie linie cyfrowe)
   WlaczLCD();          //inicjalizacja wyświetlacza LCD
   UstawKursorLCD(1, 2);       //wiersz 1. kolumna 2.
   WyswietlLCD(napis1, 11);    //wyświetl napis

   for(i=9; i>0; i--)
   {
      utoa(bufor, i, 10);    //konwersja liczby na napis
      UstawKursorLCD(1,15);    //wiersz 1. kolumna 15.
      WyswietlLCD(bufor, 1); //wyświetl napis
      __delay_ms(1000);    //zaczekaj 1 s
   }

   UstawKursorLCD(2, 4);       //wiersz 2. kolumna 4.
   WyswietlLCD(napis2, 10);    //wyświetl napis

   for(;;);          //pętla nieskończona
}
```

Obsługa serwomechanizmów

Pozostał nam do omówienia, a właściwie przypomnienia, temat obsługi serwomechanizmów. Zaraz do niego przejdziemy, odsapnijmy troszkę. Rozdział wydłużył nam się ponad miarę. Trochę jak opery Wagnera, które uwielbiam, choć czasem zastanawiam się, czy nie mogłyby być o połowę krótsze. Rossini powiedział podobno, że godzina opery Wagnera składa się z pięknych chwil i nudnych kwadransów. Nie wiem, jakie kwadranse są za nami, jeśli chodzi o ten rozdział. W każdym razie czas na piękną chwilę (patrz rysunek 2.40).

Serwomechanizm, jak pamiętamy, to nic innego jak silniczek z systemem przekładni zwiększającym jego moc i układem elektronicznym pozwalającym na sterowanie wychyleniem ramienia serwomechanizmu. Zdjęcie serwomechanizmu modelarskiego zostało przedstawione na rysunku 2.41.

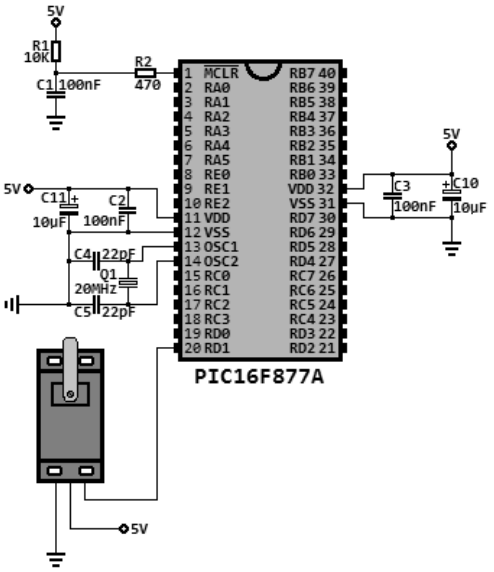
Przed nami filozoficzny dylemat: czy to z serwomechanizmu wychodzą trzy kabelki, czy trzy kabelki są dołączone do serwomechanizmu? Z którejkolwiek strony by patrzeć, kabelki są trzy: czarny, czerwony i żółty. Pierwsze dwa służą do podłączenia zasilania (czarny kabelek — masa, czerwony — źródło napięcia). Natomiast żółty kabelek podłączamy do linii mikrokontrolera. Będziemy wysyłać nim impuls służący sterowaniu serwomechanizmem. Schemat układu z serwomechanizmem widać na rysunku 2.42.



Rysunek 2.40. *Opowiastka dydaktyczna pod tytułem „Trudne i pełne wyrzeczeń życie szefa mafii"*

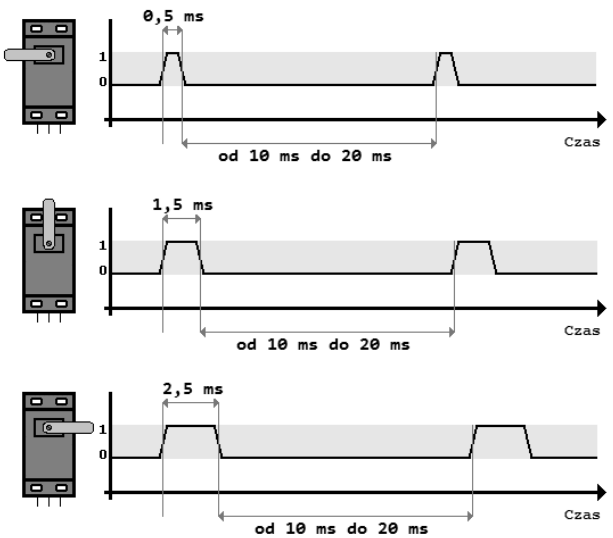
Rysunek 2.41.
Serwomechanizm HS-805BB firmy HiTEC

Zauważmy obecność w układzie dodatkowych kondensatorów C10 i C11 o pojemności 10 µF. Otóż w przypadku sterowania serwomechanizmami zachodzi niebezpieczeństwo nagłego chwilowego zaniku napięcia w obwodzie. Kondensatory mają zapobiec ewentualnemu restartowi mikrokontrolera, gdyby taki zanik napięcia się przydarzył. Ich wielkość musimy jednak dobrać do konkretnego serwomechanizmu. Aby zapobiec skutkom zaniku napięcia spowodowanego choćby niemożnością wykonania ruchu (zbyt duży ciężar), należy dodać kondensatory większe, na przykład 2200 µF. Natomiast jeśli obawiamy się zbytniego obciążenia linii RD1, możemy także między mikrokontrolerem a serwomechanizmem dać rezystor około 220 Ω. Jednak najwłaściwszą metodą budowania układu mikrokontrolera z serwomechanizmami jest podłączanie ich do różnych źródeł zasilania mających wspólną masę.



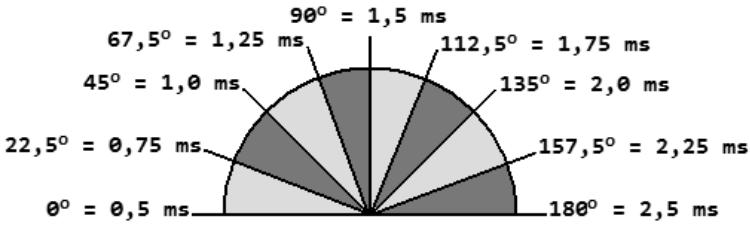
Rysunek 2.42. *Schemat układu z mikrokontrolerem PIC16F877A i podłączonym do niego serwomechanizmem*

Idźmy dalej. Ze schematu wynika, że to linii `RD1` powierzyliśmy odpowiedzialną rolę sterowania serwomechanizmem. Istotnie tak się stało. Na czym owo sterowanie polega? Otóż z linii `RD1` powinniśmy wysyłać impuls o długości określonej w dokumentacji. Najczęściej jest to impuls długości od 0,5 ms do 2,5 ms. Przerwa między impulsami powinna trwać od 10 ms do 20 ms. Ciągłe wysyłanie impulsu długości 0,5 ms skutkuje ustawieniem ramienia serwomechanizmu w skrajnej lewej pozycji. Podobnie ciągłe wysyłanie impulsu długości 2,5 ms będzie skutkowało ustawieniem ramienia serwomechanizmu w skrajnej prawej pozycji (jak wiemy, większość serwomechanizmów ma fizyczną blokadę pozwalającą na wychylenie ramienia tylko o 180°). Tę zasadę w sposób poglądowy przedstawia rysunek 2.43.



Rysunek 2.43. *Ilustracja zależności kierunku wychylenia ramienia serwomechanizmu od długości podawanego impulsu sterującego*

A co z pozostałymi kątami? Oczywiście również są do osiągnięcia dla impulsów odpowiedniej długości. Przyjmijmy, że kąt ramienia osiągany przy impulsie długości 0,5 ms jest równy 0°. Rysunek 2.44 w sposób poglądowy przedstawia kąty wychylenia ramienia serwomechanizmu osiągane przy impulsach określonej długości.



Rysunek 2.44. *Wartości wychyleń ramienia serwa otrzymane dla impulsów określonej długości*

Oto krótki przykład: aby osiągnąć pozycję środkową ramienia serwa, należy generować na linii RD1 impuls długości 1,5 ms (zakładamy podłączenie serwomechanizmu jak na rysunku 2.42). Najprostsza implementacja programowa tego zadania może wyglądać tak.

Listing R02_Prog05_C_PIC16F877A.c

```
#include <htc.h>
//definiujemy szybkość oscylatora dla funkcji __delay_
#define _XTAL_FREQ 20000000

//oscylator szybszy od 10 MHz (FOSC_HS)
//watchdog wyłączony (WDTE_OFF)
//wyłączone LVP (Low-Voltage ICSP Programming) (LVP_OFF)
__CONFIG(FOSC_HS & WDTE_OFF & LVP_OFF);

void main()
{
   ADCON1 = 0x06;           //wyłączenie linii analogowych
                            //(wszystkie linie cyfrowe)
   TRISDbits.TRISD1 = 0;        //linia RD1 wyjściowa

   for(;;)              //pętla nieskończona
   {
      PORTDbits.RD1 = 1;    //impuls na linii RD1
      __delay_us(1500);     //czekaj 1,5 ms
      PORTDbits.RD1 = 0;    //wyłącz linię RD1
      __delay_ms(18);     //czekaj 18 ms
   }
}
```

A co w przypadku, gdyby ramię serwomechanizmu miało do nas przyjaźnie machać? Jak już wspomnieliśmy, każdy impuls określonej długości należy wysyłać tak długo, aż ramię znajdzie się pod żądanym kątem. Ale uwaga! Nagła zmiana kierunku odchylania się ramienia serwomechanizmu może wywołać skoki napięcia w układzie. Pamiętajmy więc o kondensatorach, a najlepiej zapewnijmy dwa osobne źródła zasilania — jedno dla mikrokontrolera, drugie dla serwomechanizmu. A program machania ramieniem serwomechanizmu może wyglądać tak.

Listing R02_Prog06_C_PIC16F877A.c

```
#include <htc.h>
//definiujemy szybkość oscylatora dla funkcji __delay_
#define _XTAL_FREQ 20000000

//oscylator szybszy od 10 MHz (FOSC_HS)
//watchdog wyłączony (WDTE_OFF)
//wyłączone LVP (Low-Voltage ICSP Programming) (LVP_OFF)
__CONFIG(FOSC_HS & WDTE_OFF & LVP_OFF);
void main()
{
   unsigned char i, j;
   ADCON1 = 0x06;              //wyłączenie linii analogowych
                               //(wszystkie linie cyfrowe)
   TRISDbits.TRISD1 = 0;       //linia RD1 wyjściowa
   for(;;)                  //pętla nieskończona
   {
      //przesuń ramię serwomechanizmu do pozycji 0°
      for(i=0; i<80; i++)
      {
         PORTDbits.RD1 = 1; //impuls na linii RD1
         __delay_us(500);    //czekaj 0,5 ms
         PORTDbits.RD1 = 0; //wyłącz linię RD1
         __delay_ms(15);    //czekaj 15 ms
      }
     //przesuń ramię serwomechanizmu do pozycji 180°
      for(j=0; j<80; j++)
      {
         PORTDbits.RD1 = 1;    //impuls na linii RD1
         __delay_us(2500);    //czekaj 2,5 ms
         PORTDbits.RD1 = 0;    //wyłącz linię RD1
         __delay_ms(15);    //czekaj 15 ms
      }
   }
}
```

I tak niespodziewanie kończymy ten podrozdział. Nie rozwijamy tematu obsługi serwomechanizmów — wrócimy do niego w projekcie kończącym rozdział. Tymczasem zapraszam do podrozdziału dotyczącego programowania w asemblerze. To temat tylko dla prawdziwych twardzieli.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z

Ź

Mikrokontrolery PIC

w praktycznych zastosowaniach

Mikrokontrolery PIC przebojem wdarły się na rynek uniwersalnych cyfrowych układów sterujących. Obecnie są wykorzystywane do kontrolowania pracy różnych urządzeń technicznych, instalacji automatyki przemysłowej i systemów informatycznych, można je znaleźć również w używanym na co dzień sprzęcie AGD czy telefonach komórkowych. Ogromną popularność zawdzięczają sporym możliwościom, dużej niezawodności i elastyczności, prostocie programowania, szerokiemu spektrum zastosowań oraz niewygórowanym cenom.

Jedyną wadą PIC-ów wydaje się stosunkowo niewielka ilość polskojęzycznej dokumentacji, a zwłaszcza brak podręcznika, który umożliwiałby początkującemu użytkownikowi bezbolesne rozpoczęcie przygody z tymi mikrokontrolerami, zaś doświadczonemu elektronikowi — szybkie włączenie ich do swojego warsztatu pracy. Na szczęście to już przeszłość, ponieważ do rąk czytelników trafia pierwsze tak wyczerpujące kompendium wiedzy z tej dziedziny: *Mikrokontrolery PIC w praktycznych zastosowaniach*.

Niezależnie od tego, czy studiujesz robotykę, automatykę, elektronikę lub informatykę, jesteś początkującym lub zaawansowanym inżynierem, czy też technika cyfrowa to Twoje hobby i pragniesz poznać tajniki mikrokontrolerów dla własnej satysfakcji, podręcznik wprowadzi Cię w świat projektowania, konstruowania, programowania nowoczesnych mikrokomputerów jednoukładowych oraz przedstawi najistotniejsze zagadnienia związane z używaniem kilku typów PIC-ów.

Lektura nie tylko przybliży Ci podstawy asemblera i sposoby korzystania z języka C, lecz pokaże też, jak zastosować je w praktyce. Książka napisana została lekko, przystępnie i zrozumiale, stanowiąc jednocześnie naprawdę rzetelne i dogłębne kompendium wiedzy o mikrokontrolerach. To właśnie na ten podręcznik czekałeś — Twoja cierpliwość została wreszcie nagrodzona!

- Warsztat pracy programisty mikrokontrolerów PIC
- Korzystanie z programatorów, kompilatorów i IDE
- Architektura mikrokomputerów jednoukładowych PIC
- Programowanie mikrokontrolerów w asemblerze i języku C
- Obsługa urządzeń wejścia-wyjścia
- Sposoby sterowania urządzeniami zewnętrznymi
- Metody odczytywania danych zewnętrznych
- Praktyczne przykłady stosowania mikrokontrolerów PIC

Poznaj mikrokontrolery PIC od podszewki!

ISBN 978-83-246-3721-8

Cena 69,00 zł

Informatyka w najlepszym wydaniu